

Capítulo 15

Arquitetura Interna de uma UPEM

Este capítulo apresenta um sistema que integra software e hardware, idealizado pelos autores, denominado UPEM (Unidade Específica para Processamento de Periféricos de Microcontroladores) que, quando mapeado em dispositivos FPGAs, é capaz de executar funções de vários periféricos presentes em microcontroladores tradicionais. São descritos e comentados vários algoritmos de “emulação” do comportamento de periféricos, os quais foram especificamente escritos para executarem em uma UPEM que pode ser programada pelos usuários utilizando instruções de alto nível.

15.1 Especificação das UPEMs

Idealizou-se uma UPEM que possui funcionamento independente da CPU do microcontrolador. Acredita-se que essas UPEMs poderão ser adaptáveis às arquiteturas de vários outros microcontroladores, mesmo de fabricantes e arquiteturas diferentes.

Assim, ressalta-se que a proposta dos autores não é apresentada para uso exclusivo com microcontroladores PIC da Microchip. A relação com tais microcontroladores é somente justificada pela necessidade de projetar uma UPEM capaz de executar instruções assembler geradas com o uso do compilador PICBASIC.

Para executar as instruções do PICBASIC, o conjunto de instruções desta UPEM deverá ser compatível com a família PIC16X, de 14 bits.

Neste sentido, outros compiladores e microcontroladores podem ser escolhidos em trabalhos futuros, tais como o BASCOM [BASCOM, 2004], para microcontroladores compatíveis com 8051, CodeWarrior [WARRIOR, 2004], para microcontroladores HC08 etc.

Por questões de área e recursos de utilização do FPGA e futuramente de um possível ASIC, uma UPEM deve ser reduzida em relação ao PIC.

Assim, procurou-se primeiramente identificar os componentes e instruções do PIC que não são necessários para executar alguns algoritmos do PICBASIC. A principal dificuldade foi identificar esses componentes, mantendo a correta execução dos algoritmos.

- Foram descartados, previamente, todos os periféricos tradicionais presentes no PIC, tais como Captura, Timers e PWM. Os algoritmos desenvolvidos com o PICBASIC não utilizam esses periféricos e instruem a CPU principal do PIC executar algoritmos que emulam o comportamento de tais periféricos.
- Todo mecanismo de controle de interrupções e o registrador INTCON, que as habilita ou não.
- Os registradores PIE1, PIR1 e PCON, que controlam interrupções e reset.
- Prescaler, mecanismo de pull-ups e o registrador OPTION, que os controla.
- Memória EEPROM (não é possível mapear em FPGA) e USART (os comandos do PICBASIC, SERIN e SEROUT já executam funcionalidades da USART).
- Todos os registradores referentes aos Timers, ao CPP, ao Comparador A/D, EEPROM e USART.
- Inicialmente, as instruções SLEEP, que instrui ao chip entrar em modo de baixo consumo e CLRWDT, que limpa o periférico Watchdog Timer.

Com o objetivo de ilustrar algumas instruções do compilador PICBASIC que, quando executadas na CPU do PIC, realizam um comportamento similar a alguns periféricos, foi composta a tabela 15.1, na qual é possível relacionar as instruções do PICBASIC, sua função prática e o periférico equivalente que a instrução pode substituir.

Tabela 15.1 – Instruções do PICBASIC que emulam periféricos

Instrução	Função	Periférico equivalente
Pause 1000	Temporiza 1000 ms (1 segundo)	Timer
Count PORTB.1,1000,W	Conta pulsos em uma porta por 1000 ms	Captura
LCDOut \$fe, 1,"LCD"	Limpa e escreve em LCD alfanumérico	-
SerOut PORTB.0,N2400,[#W]	Envia o valor de w serialmente, à 2400 bps	USART
SerIn PORTB.6,N2400,["3"]	Espera até receber serialmente o ASCII 3	USART
PORTB.1 = 1 (ou 0)	Altera diretamente o valor na porta de saída 1	-

Estas instruções, quando trabalhadas em conjunto, possibilitam a emulação dos mais diversos periféricos ou funcionalidades necessárias em cada projeto em desenvolvimento.

15.2 Definição do Conjunto ISA

Para definir o conjunto ISA da CPU presente nas UPEMs, utilizou-se o compilador PICBASIC PRO em conjunto com o ambiente de desenvolvimento MPLAB [MPLAB, 2004].

Algumas instruções-alvo foram escritas, mantendo-se a sintaxe do PICBASIC, e, com o auxílio do MPLAB, foram verificadas as instruções assembler resultantes para cada instrução em alto nível. Esses dados estão na Tabela 15.2.

Tabela 15.2 – Instruções do BASIC, assembler e função correspondente

Instruções do PIC e mínimas declarações de variáveis para executá-las	Instruções presentes no Assembler resultante	Funcionalidade
DEFINE NO_CLRWDT 1	addwf, bcf, bsf, btfsc, btfss, call, clrf, clrw, comf, goto, incf, incfsz, iorwf, movf, movwf, return, xorwf	Conta pulsos em uma porta durante 1000 ms
VAR W WORD		Mede a frequência
COUNT PORTA.2, 1000, W		
DEFINE NO_CLRWDT 1	addlw, addwf, bcf, btfsc, btfss, call, clrf, comf, goto, incfsz, movlw, movwf, nop, return	Temporiza 1000ms
PAUSE 1000		
DEFINE NO_CLRWDT 1	andwf, bcf, bsf, btfsc, btfss, call, clrf, goto, incf, incfsz, iorwf, movf, movlw, movwf, nop, return, xorwf	Mede a largura de um pulso
larg VAR WORD		
PulsIn PORTB.6, 1, larg		
DEFINE NO_CLRWDT 1	bcf, bsf, goto	Atribui nível lógico '1' à PORTA B.6
OUTPUT PORTB.6		
PORTB.6 = 1		
DEFINE NO_CLRWDT 1	addlw, addwf, bcf, bsf, btfsc, btfss, call, clrf, comf, goto, incfsz, iorwf, movf, movlw, movwf,	Escreve em um Display LCD de caracteres
LCDOut \$fe, 1, "Escrevendo..."		
DEFINE NO_CLRWDT 1	addlw, addwf, bcf, bsf, btfsc, btfss, call, clrf, comf, decfsz, goto, incf, incfsz, iorwf, movf, movlw, nop, retlw, return, rlf, rrf, xorwf	Recebe um caractere serialmente a 2400 bps e armazena em na variável B0
Include "modedefs.bas" B0 var byte		
SerIn PORTB.0, N2400, B0		
DEFINE NO_CLRWDT 1	addlw, addwf, bcf, bsf, btfsc, btfss, call, clrf, comf, decfsz, goto, incf, incfsz, iorwf, movf, movlw, movwf, nop, return, retlw, rlf, rrf, xorwf	Envia serialmente a 2400 bps o caractere ASCII '1'
INCLUDE "modedefs.bas"		
SerOut PORTB.6, N2400, ["1"]		

Após este estudo, procurou-se definir quais seriam as instruções comuns e essenciais para a correta execução dos algoritmos. O conjunto ISA, necessário para a execução das instruções da Tabela 15.2, foi previamente estabelecido e é composto pelas instruções da Tabela 15.3.

Para executar estas instruções, foi possível uma redução no conjunto ISA presente na CPU do PIC: no PIC existem 35 instruções, e o conjunto ISA inicial para as UPEMs é de somente 25 instruções.

Tabela 15.3 – Conjunto ISA preliminar da UPEM

1 addlw	2 andwf	3 bcf	4 bsf	5 btfsf	6 btfsz	7 call	8 clrf
9 comf	10 decfsz	11 goto	12 incf	13 incfsz	14 iorwf	15 movf	16 movlw
17 movwf	18 nop	19 retlw	20 return	21 rlf	22 rrf	23 sublw	24 swapf
25 xorwf							

Contudo, verificou-se que não há sentido em escrever programas de emulação de periféricos utilizando somente as instruções em PICBASIC da Tabela 15.2. Por exemplo, tomando-se a instrução COUNT PORTA.2, 1000, W lê a frequência de um pulso na PORTA.2 e retorna o resultado em W, porém não há como realizar qualquer operação sobre a variável W.

Adicionando-se aos programas de emulação outras instruções e estruturas clássicas de programação, tais como IF THEN ELSE, WHILE END, CASE OF, REPEAT LOOP, FOR NEXT e GOTO GOSUB LABEL, também existentes no compilador PICBASIC, ampliam-se as possibilidades e a flexibilidade de cada UPEM. A Tabela 15.4 relaciona dois programas escritos em BASIC e sua respectiva funcionalidade prática.

Tabela 15.4 – Adicionando estruturas clássicas de programação

Programa	Funcionalidade prática
DEFINE NO_CLRWDT 1	Não há
VAR W WORD	
COUNT PORTA.2, 1000, W	
DEFINE NO_CLRWDT 1	Programa de emulação que indica na porta de saída PORTB.2, quando há uma frequência superior a 1Khz, na porta de entrada PORTB.1
Input PORTB.1	
Output PORTB.2	
W VAR WORD	
W = 0	
inic: Count PORTB.1,1000,W	
IF W > 1000 Then	
PORTB.2 = 1	
Else	
PORTB.2 = 0	
EndIF	
W = 0	
GoTo inic	

Um ponto positivo para este estudo é que, mesmo com a adição destas estruturas clássicas de programação (IF THEN ELSE, WHILE END, CASE OF, REPEAT LOOP, FOR NEXT, GOTO GOSUB LABEL), foi possível definir o conjunto ISA final da CPU presente nas UPEMs com apenas 29 instruções das 35 disponibilizadas no PIC. É importante ressaltar que com a adição de somente quatro instruções (addwf, andlw, decf e subwf) ao conjunto preliminar permite executar as estruturas clássicas de programação que adiciona funcionalidade prática às instruções da Tabela 15.2. Este conjunto final da CPU presente na UPEM proposta é um pouco mais detalhado na Tabela 15.5.

Tabela 15.5 – Conjunto ISA final para UPEM proposta

No.	Instrução	Funcionalidade
1	addlw	Adiciona o valor do registrador W com um literal de 8 bits (valor proveniente da memória de programa) e armazena o resultado em W.
2	addwf	Adiciona o conteúdo do registrador W com o registrador F (registrador F não existe fisicamente, são posições de memória RAM ou de programa).
3	andlw	Realiza uma operação lógica AND com o valor do registrador W e um literal de 8 bits (valor proveniente da memória de programa) e armazena o resultado em W.
4	andwf	Realiza uma operação lógica AND com o valor do registrador W e o registrador F (registrador F não existe fisicamente, são posições de memória RAM ou de programa).
5	bcf	Limpa (zera) um bit do registrador F.
6	bsf	Seta (coloca em 1) um bit do registrador F.
7	btfsc	Se bit b é "0" a próxima instrução não é executada. Se bit b é "1" a próxima instrução é executada normalmente.
8	btfss	Se bit b é "1" a próxima instrução não é executada. Se bit b é "0" a próxima instrução é executada normalmente.
9	call	Chamada de sub-rotina.
10	clrf	Registrador F é limpo (recebe 00h).
11	comf	Complementa o conteúdo do registrador F.
12	decf	Decrementa o conteúdo do registrador F.
13	decfsz	Decrementa o conteúdo do registrador F. Se o conteúdo zerar, a próxima instrução não é executada.
14	goto	Salto incondicional.
15	incf	Incrementa o conteúdo do registrador F.
16	incfsz	Incrementa o conteúdo do registrador F. Se o conteúdo zerar (rolar sobre o próprio valor), a próxima instrução não é executada.
17	iorwf	Realiza uma operação lógica OR com o valor do registrador W e o registrador F (registrador F não existe fisicamente, são posições de memória RAM ou de programa).
18	movf	O conteúdo do registrador F é movido para W ou para ele próprio, dependendo do valor de um bit presente em F.
19	movlw	Os 8 bits de uma literal k são carregadas em W.
20	movwf	Move o conteúdo do registrador W para F.
21	nop	Sem operação (apenas consome um ciclo da CPU).
22	retlw	Esta instrução auxilia no retorno de procedimentos. O registrador W é carregado com 8 bits de um literal k. O PC é carregado com o topo da pilha.
23	return	Retorno de procedimento.

Tabela 15.5 – Conjunto ISA final para UPEM proposta (cont.)

No.	Instrução	Funcionalidade
24	rif	Rotaciona o conteúdo de F, da direita para esquerda em conjunto com o bit de Carry.
25	rrf	Rotaciona o conteúdo de F, da esquerda para a direita em conjunto com o bit de Carry.
26	sublw	O conteúdo do registrador W é subtraído (complemento de 2) de 8 bits de literal k. O resultado é armazenado de volta no próprio registrador W.
27	subwf	O conteúdo do registrador W é subtraído (complemento de 2) do conteúdo de F.
28	swapf	Os bits mais significativos e os bits menos significativos (upper e lower nibbles) do registrador F são trocados.
29	xorwf	Realiza uma operação lógica XOR com o valor do registrador W e o registrador F (registrador F não existe fisicamente, são posições de memória RAM ou de programa).

O método utilizado para obtenção e análise dos opcodes necessários à execução de cada algoritmo de emulação desenvolvido, detalhado no Capítulo 13, segue o processo indicativo na Figura 15.1, o qual utiliza o software MPLAB.

Na Figura 15.1 observa-se um programa exemplo escrito em BASIC, usando o PICBASIC, e o assembler correspondente gerado que é compatível com o microcontrolador PIC. O pequeno código BASIC, do quadro em destaque à direita na mesma figura, emula as ações de um periférico de PWM, gerando uma forma de onda quadrada a 50%. De forma similar, os outros programas de emulação desenvolvidos neste estudo, utilizaram o mesmo método.

Analisando-se os opcodes gerados pelos vários programas de emulação, verificou-se que para conseguir executar os algoritmos desenvolvidos específicos para os periféricos alvo descritos neste livro (ver capítulo 12 e 13), não são necessários os seguintes opcodes: CLRW, CLRWDT, IORLW, RETFIE, SLEEP e XORLW.

```

MPLAB - C:\ARQUIV\1\MPLAB\LINK.PJT
File Project Edit Debug PICSTART Plus Options Tools Window Help
1 0000 2828 goto main
2 0001 018F PAUSE clrf 0xF
3 0002 008E PAUSE1 movwf 0xE
4 0003 30FF pause1 movlw 0xFF
5 0004 078E addwf 0xE
6 0005 1C03 btfss 0x2
7 0006 078F addwf 0xE
8 0007 1C03 btfss 0x0
9 0008 2823 goto 0xE
10 0009 3003 movlw 0
11 000A 008D movwf 0
12 000B 300F movlw 0xF
13 000C 280F call 0x0
14 000D 2803 goto 0x0
15 000E 018D PAUSEU clrf 0xF
16 000F 3EE8 PAUSEU addlw 0xF
17 0010 008C movwf 0xF
18 0011 098D comf 0xF
19 0012 30FC movlw 0xFF
20 0013 1C03 btfss 0xF
21 0014 2818 goto pause1
22 0015 078C pauseu addwf 0xF
23 0016 1803 btfsc 0x3
24 0017 2815 goto pauseu
25 0018 078C pauseu addwf 0xF
26 0019 0000 nop
27 001A 0F8D incfsz 0xF
28 001B 2815 goto pauseu
29 001C 180C btfsc 0xC, 0x0
30 001D 281E goto 0x1E
31 001E 1C8C btfss 0xC, 0x1
32 001F 2822 goto pauseudone
  
```

```

c:\arquiv\1\mplab\link.pjt
loop: PORTB_0 = 1
Pause 1
PORTB_0 = 0
Pause 1
GoTo loop
  
```

Figura 15.1 – Assembler gerado a partir de uma instrução de alto nível do PICBASIC.

15.3 Arquitetura para Cada Unidade de Processamento

Com o conjunto ISA do processador da UPEM definido, o segundo passo é definir a estrutura interna para cada UPEM. Na Figura 15.2 apresenta-se a arquitetura formada somente pelos registradores e mecanismos necessários à execução dos algoritmos do PICBASIC.

Nesta figura destacam-se duas memórias RAM: uma para armazenar os programas de emulação, os quais serão descritos na linguagem assembler da CPU da UPEM proposta (que também pode ser uma ROM ou FLASH), e outra para ser utilizada como memória de trabalho, para armazenar valores de variáveis dos programas aplicação.

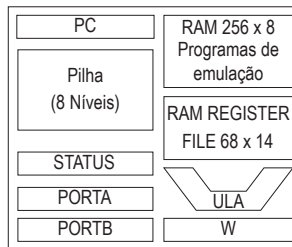


Figura 15.2 – Uma unidade de processamento compatível com o PIC.

De forma similar ao PIC, na Figura 15.2 os registradores TRISB e PORTB, juntos, formam uma estrutura de porta de E/S da UPEM, compondo oito portas bidirecionais. O registrador STATUS auxilia as operações de ULA, armazenando os bits de Carry, Zero, Underflow etc. O PC (Program Counter ou mais conhecido como Contador de Programa) é um registrador de 13 bits (não é necessário 13 bits para endereçar 256 posições, porém para manter a correta compatibilidade e evitar problemas com o assembler gerado por meio do PICBASIC, decidiu-se manter o PC com 13 bits) cuja função é endereçar a Memória de Programa.

O STACK é uma pilha com oito níveis (com a descrição VHDL, é possível adicionar mais alguns níveis, se for o caso), para armazenar o endereço de retorno do PC após uma chamada de rotina.

Conhecendo-se os trechos assembler de cada algoritmo de emulação de periféricos-alvo obtidos usando-se o PICBASIC e com a UPEM descrita em VHDL, implementada e mapeada em FPGAs, esses códigos poderão ser carregados nas memórias RAM da respectiva CPU de cada UPEM, como é ilustrado na Figura 15.3.

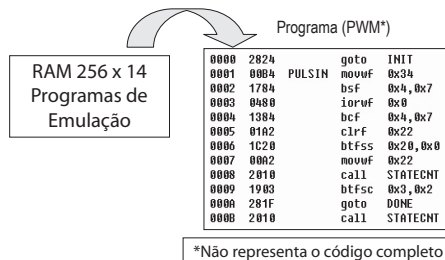


Figura 15.3 – Exemplo de código assembler de emulação de periférico carregado na RAM da UPEM.

Dentre as três novas arquiteturas apresentadas neste estudo, selecionou-se, para a descrição em VHDL e implementação em FPGAs, a Arquitetura-IV, por apresentar, teoricamente, melhorias no modo de programação.

Um possível ponto forte da Arquitetura-IV é o fato de que as UPEMs têm condições de assumir vários comportamentos diferentes, podendo substituir vários periféricos, pois as UPEMs apresentam possibilidades de funcionar em paralelo executando funções simultâneas e independentes.

Um possível ponto fraco é em relação à área final que esta Arquitetura-IV pode ocupar em um determinado FPGA. Por possuir hardware replicado, além da própria CPU principal, seria interessante investigar a área final ocupada por esta Arquitetura-IV em FPGAs. Acredita-se que, para a evolução de um projeto de um circuito integrado, é natural que ocorra um aumento na área final necessária para sua concepção.

Esta afirmação tem por base o número de transistores presente em chips que aumenta em velocidade acelerada onde no 8086, de 1985, existiam 275.000 transistores e no Itanium II, de 2003, 410.000.000 [MOORE, 2005].

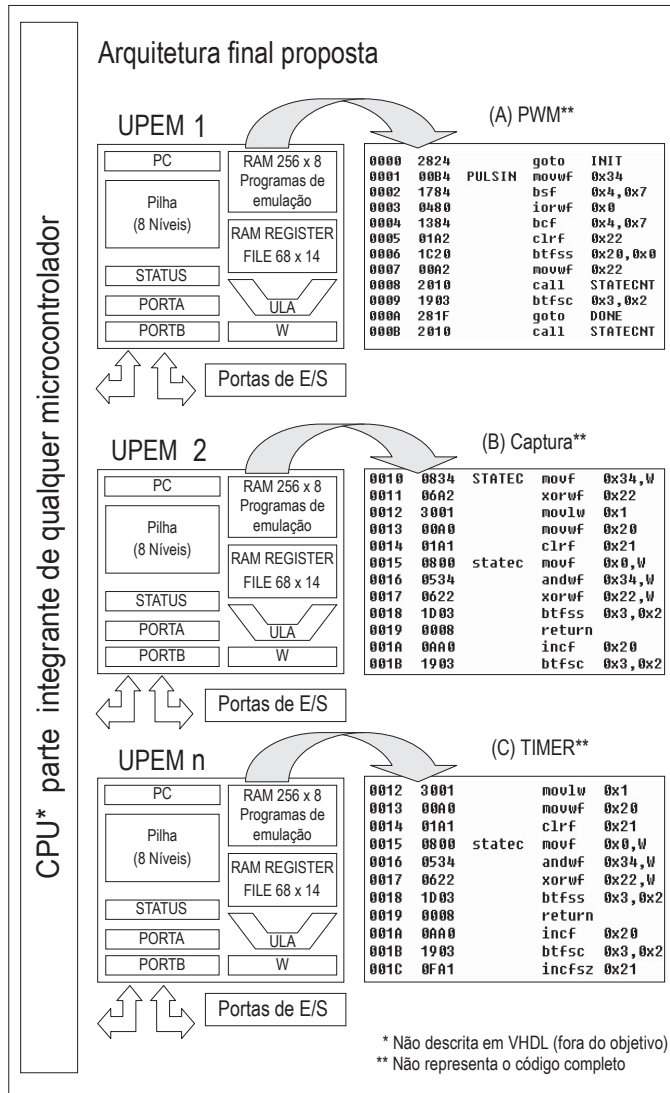
Um ponto bastante estimulante para o desenvolvimento desta Arquitetura-IV é que alguns periféricos não serão necessários, pois serão emulados nas UPEMs, resultando em diminuição da área final de ocupação do projeto no FPGA.

Na Figura 154 é representada a estrutura idealizada para a Arquitetura-IV, podendo-se observar que os códigos assembler que emulam periféricos estão carregados na memória RAM de cada UPEM. Nesta figura, parte (A), o código presente na memória RAM da primeira UPEM faz com que esta emule as ações de um periférico de PWM. De forma similar, em (B), o código da memória RAM da segunda UPEM emula um periférico de captura e em (C), o código representa um periférico temporizador.

Com a possibilidade de replicar as UPEMs, existe, então, a possibilidade de emular quantos periféricos forem necessários, sem a necessidade do desenvolvimento de Sistema Operacional Multitarefa para microcontroladores. Esse número depende da quantidade de UPEMs possíveis de serem mapeadas e sintetizadas em um determinado FPGA.

Será possível carregar o mesmo código de emulação em mais de uma memória RAM para que as UPEMs possam emular o mesmo periférico, com a possibilidade de ter várias unidades UPEM executando a mesma operação, situação comum em aplicações com microcontroladores. Por exemplo, usando somente controle de vários motores por meio de vários periféricos PWM, entre outros vários exemplos.

É importante destacar que, neste projeto, a CPU do microcontrolador presente na Arquitetura-IV não será descrita em VHDL, pois o projeto estará focado somente na possibilidade de projetar e utilizar UPEMs que tenham a capacidade de emular periféricos, investigando quais os mecanismos necessários e sua respectiva viabilidade. Assim, estuda-se a possibilidade da integração de microcontroladores com UPEMs.



- Em um trabalho futuro, de interesse de alguns leitores, a CPU do microcontrolador poderá modificar o comportamento dos periféricos emulados, trocando o código presente na memória RAM de cada UPEMs ou apenas alterando parâmetros deste.
- A execução do programa principal da automação pode ser paralela à execução dos programas nas UPEMs.

Na Figura 15.4, objetivou-se descrever em VHDL, implementar e testar em FPGAs apenas as UPEMs da Arquitetura-IV e pesquisar maneiras de extrair e validar em FPGA as seqüências de códigos assembler já existentes nos compiladores de alto nível citados anteriormente

Depois de validada a arquitetura de emulação dos periféricos (as UPEMs), poderá ser adicionada a um projeto de um microcontrolador existente, pois esta não é dependente de detalhes de funcionamento da CPU deste possível novo microcontrolador.

A CPU para a Arquitetura-IV poderá ser equivalente a qualquer CPU de qualquer outro microcontrolador, tais como os da linha TMS470, da Texas Instruments [TMS470, 2000], o SH7705, da Hitachi [SH7705, 2002], todos os compatíveis com o 8051, ou ainda qualquer outro dentre os estudados na primeira parte deste livro. Inclusive esta CPU pode ser como aquela descrita e implementada em [PENTEADO, 2002], que é compatível com o MC68HC11 da Motorola [M68HC, 2004]. Isto possibilitaria uma outra arquitetura que integra microcontroladores com periféricos em FPGAs, uma vez que simplificaria o projeto, pois este microcontrolador não precisaria dos periféricos tradicionais já existentes nos microcontroladores porque as UPEM integradas à CPU do microcontrolador podem vir a funcionar como os periféricos.

15.4 Código VHDL da CPU da UPEM

```
-- Descrição VHDL da CPU utilizada na UPEM
-- Descrição não destinada ao uso comercial.
-- Realizada com base no projeto CQPIC, de Sumio Morioka; versão original disponível em
-- http://www02.so-net.ne.jp/~morioka/cqpic.htm

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity UPEM is
port ( portbio   : inout std_logic_vector(7 downto 0);  -- Full PORT-B
      cl         : in std_logic;                       -- Clock input
      clkout2    : out std_logic;
      clkout     : out std_logic                       -- Clock output (clkIn/4) );

end UPEM;
```

```

architecture first of UPEM is

component RAM is
port( Clk      : in std_logic;
      Wr       : in std_logic;
      Addr     : in std_logic_vector (5 downto 0);
      Data_In  : in std_logic_vector (7 downto 0);
      Data_Out : out std_logic_vector(7 downto 0));
end component;

component ROM is
port ( romaddr : in std_logic_vector(12 downto 0);
      romout   : out std_logic_vector(13 downto 0));
end component;

signal ramdtin  : std_logic_vector(7 downto 0); -- RAM read data
signal ramdtout : std_logic_vector(7 downto 0); -- RAM write data
signal ramadr   : std_logic_vector(5 downto 0); -- RAM address; ramadr(8..7)
-- indicates RAM-BANK
signal writeram : std_logic; -- RAM write strobe (H active)
signal progadr  : std_logic_vector(12 downto 0); -- ROM address
signal progdata : std_logic_vector(13 downto 0); -- ROM read data
signal clkkin   : std_logic;

-- Internal registers
signal w_reg      : std_logic_vector(7 downto 0); -- W
signal pc_reg     : std_logic_vector(12 downto 0); -- PCH/PCL
signal status_reg : std_logic_vector(7 downto 0); -- STATUS
signal fsr_reg    : std_logic_vector(7 downto 0); -- FSR
signal portbin_sync_reg : std_logic_vector(7 downto 0); -- PORTB IN (synchronizer)
signal portbout_reg : std_logic_vector(7 downto 0); -- PORTB OUT
signal pclath_reg : std_logic_vector(4 downto 0); -- PCLATH
signal intcon_reg : std_logic_vector(7 downto 0); -- INTCON
signal option_reg : std_logic_vector(7 downto 0); -- OPTION
signal trisb_reg  : std_logic_vector(7 downto 0); -- TRISB

-- Internal registers for controlling instruction execution
signal inst_reg   : std_logic_vector(13 downto 0); -- Hold fetched op-code/operand
signal aluinp1_reg : std_logic_vector(7 downto 0); -- data source (1 of 2)
signal aluinp2_reg : std_logic_vector(8 downto 0); -- data source (2 of 2)
signal aluout_reg  : std_logic_vector(7 downto 0); -- result of calculation
signal exec_op_reg : std_logic; -- if L (i.e. GOTO instruction etc),
-- stall exec of instruction

-- Stack
type STACK_TYPE is array (7 downto 0) of std_logic_vector(12 downto 0);

signal stack_reg      : STACK_TYPE; -- stack body (array of data-registers)
signal stack_pnt_reg  : integer range 0 to 7; -- stack pointer (binary encoded)

```

```

signal stack_pos_node : std_logic_vector(7 downto 0); -- same with stack pointer,
-- but one-hot encoded
signal stacktop_node : std_logic_vector(12 downto 0); -- data value of stack-top

-- State register

constant Qreset : std_logic_vector(2 downto 0) := "111"; -- reset state
constant Q1 : std_logic_vector(2 downto 0) := "001"; -- state Q1
constant Q2 : std_logic_vector(2 downto 0) := "010"; -- state Q2
constant Q3 : std_logic_vector(2 downto 0) := "011"; -- state Q3
constant Q4 : std_logic_vector(2 downto 0) := "100"; -- state Q4
signal state_reg : std_logic_vector(2 downto 0) := Qreset;

signal INST_ADDLW, INST_ADDWF, INST_ANDLW, INST_ANDWF, INST_BCF, INST_BSF,
INST_BTFSC, INST_BTFSS, INST_CALL, INST_CLRF, INST_CLRW, INST_COMF, INST_DECF,
INST_DECFSZ, INST_GOTO, INST_INCF, INST_INCFSZ, INST_IORLW, INST_IORWF,
INST_MOVLW, INST_MOVF, INST_MOVWF, INST_RETFIE, INST_RETLW, INST_RET,
INST_RLF, INST_RRF, INST_SUBLW, INST_SUBWF, INST_SWAPF, INST_XORLW, INST_XORWF:
std_logic;

signal ramadr_node : std_logic_vector(8 downto 0); -- RAM access address

signal ADDR_PCL, ADDR_STAT, ADDR_FSR, ADDR_PORTB, ADDR_PCLATH, ADDR_INTCON,
ADDR_OPTION, ADDR_TRISB, ADDR_SRAM: std_logic;

signal writeram_reg : std_logic; -- data-sram write strobe
signal clkout_reg : std_logic; -- clkout output

-- *****
-- Sinais necessários para calibrar o clock. O Pic e os programas escritos em
-- PicBasic, operam em clock de 4 Mhz. A placa de testes com FPGA (DI02 e D2E)
-- utiliza um clock base de 50MHz.
-- Dividindo-se o clock de 50MHz, não é possível obter exatamente 4MHz.
-- O valor mais próximo alcançado foi 4166MHz. Foi necessário incluir este
-- circuito que calibra o clock, obtendo-se 4 milhões de pulsos por segundo
-- (que representa um clock de 4MHz). Caso o projeto seja implementado em uma
-- placa onde é possível gerar diretamente 4MHz, desconsidere estes processos

signal CNT4166 : integer range 0 to 50;
signal C1,C2 : integer range 0 to 101:= 0;
signal T1 : std_logic:='1';
signal T2,CL4166 : std_logic:='0';
-- *****

begin
  Process(CL)
  Begin
    if CL'event and CL='1' then
      CNT4166 <= CNT4166+1;
      If CNT4166 <= 5 then
        CL4166 <= '1';

```

```

else
    CL4166 <= '0';
end if;
If CNT4166 = 11 then
    CNT4166 <= 0 ;
end if;
end if;
End process;
-- *****
-- Para calibrar o clock, comente (ou apague os caracteres de comentários "--")
-- na comparação realizada no comando IF. É necessário que os dois processos tenham
-- os mesmos parâmetros para C1 e C2 (0, 10, 20, 30...etc) comentados ou não.

Process(CL4166)
Begin
if CL4166'event and CL4166='1' then
C1 <= C1+1;
If (C1 /= 0) and (C1 /= 10)
-- and (C1 /= 20)
    and (C1 /= 30)
-- and (C1 /= 40)
-- and (C1 /= 50)
-- and (C1 /= 60)
-- and (C1 /= 70)
-- and (C1 /= 80)
    and (C1 /= 90) then
    T2 <= not T2;
end if;
If C1 = 100 then
    C1 <= 0 ;
end if;
end if;
End process;

Process(CL4166)
Begin
if CL4166'event and CL4166='0' then
C2 <= C2+1;
If (C2 /= 0) and (C2 /= 10)
-- and (C2 /= 20)
    and (C2 /= 30)
-- and (C2 /= 40)
-- and (C2 /= 50)
-- and (C2 /= 60)
-- and (C2 /= 70)
-- and (C2 /= 80)
and (C2 /= 90) then
    T1 <= not T1;
end if;

```

```

If C2 = 100 then
    C2 <= 0;
end if;
end if;
End process;

```

```

CLKIN <= T1 xor T2;          -- 4 milhões de pulsos por segundo,
                             -- a partir de um clock de 50MHz

```

```
-- *****
```

```

INST_CALL <= '1' when inst_reg(13 downto 11) = "100"   else '0';
INST_GOTO <= '1' when inst_reg(13 downto 11) = "101"   else '0';
INST_BCF  <= '1' when inst_reg(13 downto 10) = "0100"  else '0';
INST_BSF  <= '1' when inst_reg(13 downto 10) = "0101"  else '0';
INST_BTFSC <= '1' when inst_reg(13 downto 10) = "0110"  else '0';
INST_BTFSS <= '1' when inst_reg(13 downto 10) = "0111"  else '0';
INST_MOVLW <= '1' when inst_reg(13 downto 10) = "1100"  else '0';
INST_RETLW <= '1' when inst_reg(13 downto 10) = "1101"  else '0';
INST_SUBLW <= '1' when inst_reg(13 downto 9) = "11110"  else '0';
INST_ADDLW <= '1' when inst_reg(13 downto 9) = "11111"  else '0';
INST_IORLW <= '1' when inst_reg(13 downto 8) = "111000"  else '0';
INST_ANDLW <= '1' when inst_reg(13 downto 8) = "111001"  else '0';
INST_XORLW <= '1' when inst_reg(13 downto 8) = "111010"  else '0';
INST_SUBWF <= '1' when inst_reg(13 downto 8) = "000010"  else '0';
INST_DECF <= '1' when inst_reg(13 downto 8) = "000011"  else '0';
INST_IORWF <= '1' when inst_reg(13 downto 8) = "000100"  else '0';
INST_ANDWF <= '1' when inst_reg(13 downto 8) = "000101"  else '0';
INST_XORWF <= '1' when inst_reg(13 downto 8) = "000110"  else '0';
INST_ADDWF <= '1' when inst_reg(13 downto 8) = "000111"  else '0';
INST_MOVF <= '1' when inst_reg(13 downto 8) = "001000"  else '0';
INST_COMF <= '1' when inst_reg(13 downto 8) = "001001"  else '0';
INST_INCF <= '1' when inst_reg(13 downto 8) = "001010"  else '0';
INST_DECFSZ <= '1' when inst_reg(13 downto 8) = "001011"  else '0';
INST_RRF  <= '1' when inst_reg(13 downto 8) = "001100"  else '0';
INST_RLF  <= '1' when inst_reg(13 downto 8) = "001101"  else '0';
INST_SWAPF <= '1' when inst_reg(13 downto 8) = "001110"  else '0';
INST_INCFSZ <= '1' when inst_reg(13 downto 8) = "001111"  else '0';
INST_MOVWF <= '1' when inst_reg(13 downto 7) = "0000001" else '0';
INST_CLRW <= '1' when inst_reg(13 downto 7) = "0000010" else '0';
INST_CLRF <= '1' when inst_reg(13 downto 7) = "0000011" else '0';
INST_RET  <= '1' when inst_reg(13 downto 0) = "00000000001000" else '0';
INST_RETFIE <= '1' when inst_reg(13 downto 0) = "00000000001001" else '0';
--INST_CLRWDT <= '1' when inst_reg(13 downto 0) = "00000001100100" else '0';

```

```

ramadr_node <= status_reg(7) & fsr_reg(7 downto 0) when inst_reg(6 downto 0) =
"0000000" else

```

```
status_reg(6 downto 5) & inst_reg(6 downto 0);
```

```
ADDR_SRAM <= '1' when ramadr_node(6 downto 0) > "0001011" else '0'; -- 0CH-7FH, 8CH-FFH
```

```

ADDR_PCL  <= '1' when ramadr_node(6 downto 0) = "0000010" else '0'; -- 02H, 82H
ADDR_STAT <= '1' when ramadr_node(6 downto 0) = "0000011" else '0'; -- 03H, 83H
ADDR_FSR  <= '1' when ramadr_node(6 downto 0) = "0000100" else '0'; -- 04H, 84H
ADDR_PORTB <= '1' when ramadr_node(7 downto 0) = "00000110" else '0'; -- 06H
ADDR_PCLATH <= '1' when ramadr_node(6 downto 0) = "0001010" else '0'; -- 0AH, 8AH
ADDR_INTCON <= '1' when ramadr_node(6 downto 0) = "0001011" else '0'; -- 0BH, 8BH
ADDR_OPTION <= '1' when ramadr_node(7 downto 0) = "10000001" else '0'; -- 81H
ADDR_TRISB <= '1' when ramadr_node(7 downto 0) = "10000110" else '0'; -- 86H

```

```

ND1: for I in 0 to 7 generate
stack_pos_node(I) <= '1' when stack_pnt_reg = I else '0';
end generate ND1;

```

```

u2:process (stack_reg, stack_pos_node)
variable stack_cell : STACK_TYPE; -- value of each stack cell
variable top : std_logic_vector(12 downto 0); -- value of stack top
begin
for I in 0 to 7 loop
if (stack_pos_node(I) = '1') then -- (if the position is stack top)
stack_cell(I) := stack_reg(I);
else
stack_cell(I) := "00000000000000";
end if;
end loop;

top := stack_cell(0);
for I in 1 to 7 loop
top := top or stack_cell(I);
end loop;
stacktop_node <= top;
end process;

```

```

u3:process (clkin)
variable ramin_node : std_logic_vector(7 downto 0); -- result of reading RAM/
-- Special registers
variable incpc_node : std_logic_vector(12 downto 0); -- value of PC + 1
variable mask_node : std_logic_vector(7 downto 0); -- bit mask for logical
-- operations
variable add_node : std_logic_vector(8 downto 0); -- result of 8bit addition
-- (std_logic_vector)
variable addLow_node : std_logic_vector(4 downto 0); -- result of low-4bit
-- addition (std_logic_vector)
variable aluout_zero_node : std_logic; -- H if ALUOUT = 0
variable writew_node : std_logic; -- H if destination is W register
variable writeram_node : std_logic; -- H if destination is RAM/Special
-- registers

variable stack_full_node : integer range 0 to 7;
variable extbit_node : std_logic;

```

```

begin
if (c1kin'event and c1kin = '1') then
if (ADDR_SRAM = '1') then
ramin_node := ramdtin;           -- data bus output of external SRAM
elsif (ADDR_PCL = '1') then
ramin_node := pc_reg(7 downto 0); -- PCL
elsif (ADDR_STAT = '1') then
ramin_node := status_reg;       -- STATUS
elsif (ADDR_FSR = '1') then
ramin_node := fsr_reg;          -- FSR
elsif (ADDR_PORTB = '1') then
for I in 0 to 7 loop
if (trisb_reg(I) = '1') then
ramin_node(I) := portbin_sync_reg(I); -- PORT B (when input mode)
else
ramin_node(I) := portbout_reg(I);    -- PORT B (when output mode)
end if;
end loop;
elsif (ADDR_PCLATH = '1') then
ramin_node := "000" & pclath_reg; -- PCLATH (5bit)
elsif (ADDR_INTCON = '1') then
ramin_node := intcon_reg;        -- INTCON
elsif (ADDR_OPTION = '1') then
ramin_node := option_reg;       -- OPTION
elsif (ADDR_TRISB = '1') then
ramin_node := trisb_reg;       -- TRISB
else
ramin_node := (others => '0');
end if;
incpc_node := pc_reg + "0000000000001";
add_node := ("0" & aluinp1_reg) + aluinp2_reg;
if (INST_SUBLW = '1' or INST_SUBWF = '1') then
extbit_node := aluinp2_reg(4);
else
extbit_node := '0';
end if;
addLow_node := ("0" & aluinp1_reg(3 downto 0)) + (extbit_node & aluinp2_reg(3
downto 0));
if (aluout_reg = "00000000") then aluout_zero_node := '1'; else
aluout_zero_node := '0';
end if;

if (INST_MOVWF = '1' or INST_BCF = '1' or INST_BSF = '1' or INST_CLRF = '1') then
writew_node := '0';
writeram_node := '1';
elsif (INST_MOVLW = '1' or INST_ADDLW = '1' or INST_SUBLW = '1' or INST_ANDLW = '1'
or INST_IORLW = '1'
or INST_XORLW = '1' or INST_RETLW = '1' or INST_CLRW = '1') then
writew_node := '1';
writeram_node := '0';

```

```

elsif (INST_MOVF = '1' or INST_SWAPF = '1' or INST_ADDWF = '1' or INST_SUBWF = '1'
or INST_ANDWF = '1' or INST_IORWF = '1' or INST_XORWF = '1' or INST_DECF = '1' or
INST_INCF = '1' or INST_RLF = '1'
or INST_RRF = '1' or INST_DECFSZ = '1' or INST_INCFSZ = '1' or INST_COMF = '1') then
writew_node := not inst_reg(7);      -- ("d" field of fetched instruction)
writeram_node := inst_reg(7);      -- ("d" field of fetched instruction)
else
writew_node := '0';
writeram_node := '0';
end if;
-- 2. EFSM body
case state_reg is
when Qreset =>
pc_reg <= (others => '0'); -- 0
status_reg <= "00011000"; -- Valor Power On Reset
pclath_reg <= (others => '0'); -- 0
intcon_reg(7 downto 1) <= "00000000";
option_reg <= (others => '1');
trisb_reg <= (others => '1');
exec_op_reg <= '0';
writeram_reg <= '0';
state_reg <= Q1;
when Q1 => -- portbin_sync_reg <= portb_in;
--portbin_sync_reg <= portbio;
if trisb_reg(0) = '1' then
portbin_sync_reg(0) <= portbio(0);
else portbin_sync_reg(0) <= 'Z'; end if;
if trisb_reg(1) = '1' then
portbin_sync_reg(1) <= portbio(1);
else portbin_sync_reg(1) <= 'Z'; end if;
if trisb_reg(2) = '1' then
portbin_sync_reg(2) <= portbio(2);
else portbin_sync_reg(2) <= 'Z'; end if;
if trisb_reg(3) = '1' then
portbin_sync_reg(3) <= portbio(3);
else portbin_sync_reg(3) <= 'Z'; end if;
if trisb_reg(4) = '1' then
portbin_sync_reg(4) <= portbio(4);
else portbin_sync_reg(4) <= 'Z'; end if;
if trisb_reg(5) = '1' then
portbin_sync_reg(5) <= portbio(5);
else portbin_sync_reg(5) <= 'Z'; end if;
if trisb_reg(6) = '1' then
portbin_sync_reg(6) <= portbio(6);
else portbin_sync_reg(6) <= 'Z'; end if;
if trisb_reg(7) = '1' then
portbin_sync_reg(7) <= portbio(7);
else portbin_sync_reg(7) <= 'Z'; end if;

state_reg <= Q2;

```

```

when Q2 => if exec_op_reg = '1' then
-- Set aluinp1 register (source #1)
if (INST_MOVF = '1' or INST_SWAPF = '1' or INST_ADDWF = '1' or INST_SUBWF = '1'
    or INST_ANDWF = '1' or INST_IORWF = '1' or INST_XORWF = '1' or INST_DECF = '1'
    or INST_INCF = '1' or INST_RLF = '1' or INST_RRF = '1' or INST_BCF = '1'
    or INST_BSF = '1' or INST_BTFSC = '1' or INST_BTFSS = '1' or INST_DECFSZ = '1'
    or INST_INCFSZ = '1' or INST_COMF = '1') then
    aluinp1_reg <= ramin_node;           -- RAM/Special registers
elsif (INST_MOVLW = '1' or INST_ADDLW = '1' or INST_SUBLW = '1' or INST_ANDLW = '1'
    or INST_IORLW = '1' or INST_XORLW = '1' or INST_RETLW = '1') then
    aluinp1_reg <= inst_reg(7 downto 0); -- Immediate value ("k")
elsif (INST_CLRF = '1' or INST_CLRW = '1') then
    aluinp1_reg <= (others => '0');      -- 0
else
    aluinp1_reg <= w_reg;               -- W register
end if;

-- Set aluinp2 register (source #2)
case inst_reg(9 downto 7) is
-- construct bit-mask for
-- logical operations/bit test
    when "000" => mask_node := "00000001";
    when "001" => mask_node := "00000010";
    when "010" => mask_node := "00000100";
    when "011" => mask_node := "00001000";
    when "100" => mask_node := "00010000";
    when "101" => mask_node := "00100000";
    when "110" => mask_node := "01000000";
    when others => mask_node := "10000000";
end case;

if (INST_DECF = '1' or INST_DECFSZ = '1') then
    aluinp2_reg <= (others => '1');      -- -1 (for decrement)
elsif (INST_INCF = '1' or INST_INCFSZ = '1') then
    aluinp2_reg <= "000000001";       -- 1 (for increment)
elsif (INST_SUBLW = '1' or INST_SUBWF = '1') then
    aluinp2_reg <= ("1" & (not w_reg)) + "000000001"; -- -1 * W register
--(for subtract)
elsif (INST_BCF = '1') then
    aluinp2_reg(8) <= '0';
    aluinp2_reg(7 downto 0) <= not (mask_node); -- mask for BCF: value of
-- only one position is '0'
    -- operation of BCF: AND with inverted mask ("1..101..1")
elsif (INST_BTFSC = '1' or INST_BTFSS = '1' or INST_BSF = '1') then
    aluinp2_reg <= "0" & mask_node;    -- operation of BSF: OR with
-- mask_node ("0..010..0")
    -- operation of FSC and FSS: AND with mask_node and then compare with zero
else
    aluinp2_reg <= "0" & w_reg;        -- W register
end if;
if (INST_RET = '1' or INST_RETLW = '1' or INST_RETFIE = '1') then

```

```

    if (stack_pnt_reg = 0) then
        stack_pnt_reg <= 7;           -- if pointer=0, then next
                                     -- value should be STACK_SIZE-1
    else
        stack_pnt_reg <= stack_pnt_reg - 1;  -- otherwise, current value - 1
    end if;
end if;
clkout_reg <= '1';
state_reg <= Q3;
when Q3 => -- Calculation and store result into alu-output register
    if exec_op_reg = '1' then        -- if NOT STALLED
        -- Set aluout register
        if (INST_RLF = '1') then
            aluout_reg <= aluinp1_reg(6 downto 0) & status_reg(0);    -- rotate
left
            elsif (INST_RRF = '1') then
                aluout_reg <= status_reg(0) & aluinp1_reg(7 downto 1);    -- rotate
right
            elsif (INST_SWAPF = '1') then
                -- swap H-nibble and L-nibble
                aluout_reg <= aluinp1_reg(3 downto 0) & aluinp1_reg(7 downto 4);
            elsif (INST_COMF = '1') then
                aluout_reg <= not aluinp1_reg;    -- logical inversion
            elsif (INST_ANDLW = '1' or INST_ADDWF = '1' or INST_BCF = '1'
                or INST_BTFC = '1' or INST_BTFS = '1') then
                aluout_reg <= aluinp1_reg and aluinp2_reg(7 downto 0);    -- logical AND/
bit clear/bit test
            elsif (INST_BSF = '1' or INST_IORLW = '1' or INST_IORWF = '1') then
                aluout_reg <= aluinp1_reg or aluinp2_reg(7 downto 0);    -- logical OR/bit
set
            elsif (INST_XORLW = '1' or INST_XORWF = '1') then
                aluout_reg <= aluinp1_reg xor aluinp2_reg(7 downto 0);    -- logical XOR
            elsif (INST_ADDLW = '1' or INST_ADDWF = '1' or INST_SUBLW = '1' or INST_
SUBWF = '1'
                or INST_DECF = '1' or INST_DECFSZ = '1' or INST_INCF = '1' or INST_INCFSZ
= '1') then
                aluout_reg <= add_node(7 downto 0);    -- addition/subtraction/
increment/decrement
            else
                aluout_reg <= aluinp1_reg;    -- pass through
            end if;
        -- Set C flag and DC flag
        if (INST_ADDLW = '1' or INST_ADDWF = '1') then
            status_reg(1) <= addLow_node(4);    -- DC flag
            status_reg(0) <= add_node(8);    -- C flag
        elsif (INST_SUBLW = '1' or INST_SUBWF = '1') then
            status_reg(1) <= not addLow_node(4);    -- DC flag
            status_reg(0) <= not add_node(8);    -- C flag
        elsif (INST_RLF = '1') then
            status_reg(0) <= aluinp1_reg(7);    -- C flag

```

```

elsif (INST_RRF = '1') then
    status_reg(0) <= aluinp1_reg(0); -- C flag
end if;

-- Set data-SRAM write enable (hazard-free)
if (writeram_node = '1' and ADDR_SRAM = '1') then
    writeram_reg <= '1';
else
    writeram_reg <= '0';
end if;

else writeram_reg <= '0'; end if;
state_reg <= Q4;

when Q4 => -- Fetch next program-instruction
    inst_reg <= progdata;
    if exec_op_reg = '0' then
        pc_reg <= incpc_node; -- increment PC
        exec_op_reg <= '1';
    else
-- Set W register
        if (writew_node = '1') then w_reg <= aluout_reg; end if;

-- Set data RAM/special registers
        if (writeram_node = '1') then
            if (ADDR_STAT = '1') then
                status_reg(7 downto 5) <= aluout_reg(7 downto 5); -- write
IRP,RP1,RP0
                status_reg(1 downto 0) <= aluout_reg(1 downto 0); -- write DC,C
            end if;
            if (ADDR_FSR = '1') then
                fsr_reg <= aluout_reg; -- write FSR
            end if;
            if (ADDR_PORTB = '1') then
                portbout_reg <= aluout_reg; -- write PORT-B
            end if;
            if (ADDR_PCLATH = '1') then
                pclath_reg <= aluout_reg(4 downto 0); -- write PCLATH
            end if;
            if (ADDR_INTCON = '1') then
                intcon_reg(6 downto 0) <= aluout_reg(6 downto 0); -- write INTCON
(except GIE)
            end if;
            if (ADDR_OPTION = '1') then
                option_reg <= aluout_reg; -- write OPTION
            end if;
            if (ADDR_TRISB = '1') then
                trisb_reg <= aluout_reg; -- write TRISB
            end if;
        end if;
    end if;

```

```

-- Set/clear Z flag, if not in stall cycle (intstart_reg = '0')
  if (ADDR_STAT = '1') then
    status_reg(2) <= aluout_reg(2);    -- (distination is Z flag)
  elsif (INST_ADDLW = '1' or INST_ADDWF = '1' or INST_ANDLW = '1' or INST_ANDWF =
'1'
    or INST_CLRF = '1' or INST_CLRW = '1' or INST_COMF = '1' or INST_DECF = '1'
    or INST_INCF = '1' or INST_MOVF = '1' or INST_SUBLW = '1' or INST_SUBWF =
'1'
    or INST_XORLW = '1' or INST_XORWF = '1') then
    status_reg(2) <= aluout_zero_node; -- Z=1 if result == 0
  elsif (INST_IORLW = '1' or INST_IORWF = '1') then

    -- IMPORTANTE : SELECIONE UMA DAS SEGUINTE LINHAS DE CÓDIGO

    -- status_reg(2) <= not aluout_zero_node;
    status_reg(2) <= aluout_zero_node;
  end if;
-- Set PC register and determine if execute next cycle or not
  if (INST_RET = '1' or INST_RETLW = '1' or INST_RETFIE = '1') then
    pc_reg <= stacktop_node;
    exec_op_reg <= '0';
  elsif (INST_GOTO = '1' or INST_CALL = '1') then
    pc_reg <= "00" & inst_reg(10 downto 0);
    exec_op_reg <= '0';
  elsif ( ((INST_BTFSZ = '1' or INST_DECFSZ = '1' or INST_INCFSZ = '1') and
aluout_zero_node = '1')
    or (INST_BTFSZ = '1' and aluout_zero_node = '0') ) then -- bit_test
instructions
    pc_reg <= incpc_node;
    exec_op_reg <= '0';
  elsif (writeram_node = '1' and ADDR_PCL = '1') then
    pc_reg <= pclath_reg(4 downto 0) & aluout_reg;
    exec_op_reg <= '0';
  else
    if (INST_GOTO = '0' or INST_CALL = '0') then
      pc_reg <= incpc_node;
      exec_op_reg <= '1';
    end if;
  end if;
  if INST_CALL = '1' then
    -- write PC-value into stack top
    for I in 0 to 7 loop
      if (stack_pos_node(I) = '1') then -- check if the stack cell is stack
top or not
        stack_reg(I) <= pc_reg; -- if so, write PC value
      end if;
    end loop;
    -- increment stack pointer
    stack_full_node := 7;
    if (stack_pnt_reg = stack_full_node) then

```

```

        stack_pnt_reg <= 0;
    else
        stack_pnt_reg <= stack_pnt_reg + 1;
    end if;
end if;

end if;
    writeram_reg <= '0';
    clkout_reg <= '0';
    state_reg <= Q1;
-- Illegal state
when others => state_reg <= Qreset;
end case;
end if;
end process;

-- data RAM data bus/address bus/control signals
U4: RAM port map ( Clk => clkin, Wr => writeram, Addr => ramadr, Data_In =>
ramdtout, Data_Out => ramdtin);

-- program ROM data bus/address bus
progadr <= pc_reg;
U5: ROM port map (romaddr => progadr, romout=> progdata );
ramadr <= ramadr_node(5 downto 0); -- data RAM address
ramdtout <= aluout_reg; -- data RAM write data
writeram <= writeram_reg; -- data RAM write enable

portbio(0) <= portbout_reg(0) when trisb_reg(0) = '0' else 'Z';
portbio(1) <= portbout_reg(1) when trisb_reg(1) = '0' else 'Z';
portbio(2) <= portbout_reg(2) when trisb_reg(2) = '0' else 'Z';
portbio(3) <= portbout_reg(3) when trisb_reg(3) = '0' else 'Z';
portbio(4) <= portbout_reg(4) when trisb_reg(4) = '0' else 'Z';
portbio(5) <= portbout_reg(5) when trisb_reg(5) = '0' else 'Z';
portbio(6) <= portbout_reg(6) when trisb_reg(6) = '0' else 'Z';
portbio(7) <= portbout_reg(7) when trisb_reg(7) = '0' else 'Z';

    clkout <= not clkout_reg; -- clkout (clk/4) output
    clkout2  <= not clkin;
end;

-- Descrição da memória RAM da UPEM

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RAM is
    port(
        Clk: in std_logic;
        Wr : in std_logic;

```

```

    Addr  : in std_logic_vector(5 downto 0);
    Data_In  : in std_logic_vector(7 downto 0);
    Data_Out : out std_logic_vector(7 downto 0)
  );
end RAM;

```

architecture rtl of RAM is

```

    type RAM_Image is array (63 downto 0) of std_logic_vector(7 downto 0);
    signal RAM : RAM_Image;
    signal AddrRd : std_logic_vector(5 downto 0);
    signal AddrWr : std_logic_vector(5 downto 0);
    Signal first : boolean:= true;
begin

    process (Clk)
    begin
        if Clk'event and Clk = '1' then
            if first then
                for i in 0 to 63 loop
                    RAM (i) <= (others=>'0');
                end loop;
                first <= false;
            end if;
            AddrRd <= Addr;
            AddrWr <= Addr;
            if Wr = '1' then
                RAM(to_integer(unsigned(AddrWr))) <= Data_In;
            end if;
        end if;
    end process;

    Data_Out <= RAM(to_integer(unsigned(AddrRd)));

end;
-- Device xc2s200e-ft256-7 utilization summary:
-- Number of External GCLKIOBs 1 out of 4 25%
-- Number of External IOBs 23 out of 178 12%
-- Number of LOCed External IOBs 0 out of 23 0%
-- Number of SLICES 492 out of 2352 20%
-- Number of GCLKs 1 out of 4 25%

-- VHDL (ROM) - Um dos programas desenvolvidos para uma UPEM
-- Gera 3 pulsos distintos de PWM na PORTb.7
-- PWM PORTB.7, 10, 50 e 90% 2 Khz
library ieee;
use ieee.std_logic_1164.all;

entity rom is

```

```

port (
    romaddr  : in std_logic_vector(12 downto 0);
    romout   : out std_logic_vector(13 downto 0)
);
end rom;

architecture RTL of rom is
begin
    process (romaddr)
    begin
        case romaddr is
            when "000000000000" => romout <= "10100000101110"; -- addr 0000: data 282E
            when "000000000001" => romout <= "00000110001101"; -- addr 0001: data 018D
            when "000000000010" => romout <= "11111011101000"; -- addr 0002: data 3EE8
            when "000000000011" => romout <= "00000010001100"; -- addr 0003: data 008C
            when "000000000100" => romout <= "00100110001101"; -- addr 0004: data 098D
            when "000000000101" => romout <= "11000011111100"; -- addr 0005: data 30FC
            when "000000000110" => romout <= "01110000000011"; -- addr 0006: data 1C03
            when "000000000111" => romout <= "10100000001011"; -- addr 0007: data 280B
            when "000000001000" => romout <= "00011110001100"; -- addr 0008: data 078C
            when "000000001001" => romout <= "01100000000011"; -- addr 0009: data 1803
            when "000000001010" => romout <= "10100000001000"; -- addr 000A: data 2808
            when "000000001011" => romout <= "00011110001100"; -- addr 000B: data 078C
            when "000000001100" => romout <= "00000000000000"; -- addr 000C: data 0000
            when "000000001101" => romout <= "00111110001101"; -- addr 000D: data 0F8D
            when "000000001110" => romout <= "10100000001000"; -- addr 000E: data 2808
            when "000000001111" => romout <= "01100000001100"; -- addr 000F: data 180C
            when "0000000010000" => romout <= "10100000010001"; -- addr 0010: data 2811
            when "0000000010001" => romout <= "01110010001100"; -- addr 0011: data 1C8C
            when "0000000010010" => romout <= "10100000010101"; -- addr 0012: data 2815
            when "0000000010011" => romout <= "00000000000000"; -- addr 0013: data 0000
            when "0000000010100" => romout <= "10100000010101"; -- addr 0014: data 2815
            when "0000000010101" => romout <= "00000000001000"; -- addr 0015: data 0008
            when "0000000010110" => romout <= "00000010001110"; -- addr 0016: data 008E
            when "0000000010111" => romout <= "11000000000001"; -- addr 0017: data 3001
            when "0000000011000" => romout <= "10100000011001"; -- addr 0018: data 2819
            when "0000000011001" => romout <= "00000010010100"; -- addr 0019: data 0094
            when "0000000011010" => romout <= "00100000001111"; -- addr 001A: data 080F
            when "0000000011011" => romout <= "00001000001101"; -- addr 001B: data 020D
            when "0000000011100" => romout <= "01110100000011"; -- addr 001C: data 1D03
            when "0000000011101" => romout <= "10100000100000"; -- addr 001D: data 2820
            when "0000000011110" => romout <= "00100000001110"; -- addr 001E: data 080E
            when "0000000011111" => romout <= "00001000001100"; -- addr 001F: data 020C
            when "0000000100000" => romout <= "110000000000100"; -- addr 0020: data 3004
            when "0000000100001" => romout <= "01100000000011"; -- addr 0021: data 1803
            when "0000000100010" => romout <= "11000000000001"; -- addr 0022: data 3001
            when "0000000100011" => romout <= "01100100000011"; -- addr 0023: data 1903
            when "0000000100100" => romout <= "11000000000010"; -- addr 0024: data 3002
            when "0000000100101" => romout <= "00010100010100"; -- addr 0025: data 0514
        end case;
    end process;
end architecture;

```

```

when "000000100110" => romout <= "01110100000011"; -- addr 0026: data 1D03
when "000000100111" => romout <= "11000011111111"; -- addr 0027: data 30FF
when "000000101000" => romout <= "10100000101001"; -- addr 0028: data 2829
when "000000101001" => romout <= "01001110000011"; -- addr 0029: data 1383
when "000000101010" => romout <= "01001100000011"; -- addr 002A: data 1303
when "000000101011" => romout <= "01001010000011"; -- addr 002B: data 1283
when "000000101100" => romout <= "00000000000000"; -- addr 002C: data 0000
when "000000101101" => romout <= "00000000001000"; -- addr 002D: data 0008
when "000000101110" => romout <= "01011010000011"; -- addr 002E: data 1683
when "000000101111" => romout <= "01001110000110"; -- addr 002F: data 1386
when "000000110000" => romout <= "01001010000011"; -- addr 0030: data 1283
when "000000110001" => romout <= "00000110100100"; -- addr 0031: data 01A4
when "000000110010" => romout <= "00000110100101"; -- addr 0032: data 01A5
when "000000110011" => romout <= "00100000100100"; -- addr 0033: data 0824
when "000000110100" => romout <= "00000010001100"; -- addr 0034: data 008C
when "000000110101" => romout <= "00100000100101"; -- addr 0035: data 0825
when "000000110110" => romout <= "00000010001101"; -- addr 0036: data 008D
when "000000110111" => romout <= "11000000010011"; -- addr 0037: data 3013
when "000000111000" => romout <= "00000010001111"; -- addr 0038: data 008F
when "000000111001" => romout <= "11000010001000"; -- addr 0039: data 3088
when "000000111010" => romout <= "10000000010110"; -- addr 003A: data 2016
when "000000111011" => romout <= "01110100000011"; -- addr 003B: data 1D03
when "000000111100" => romout <= "10100001000010"; -- addr 003C: data 2842
when "000000111101" => romout <= "10000001100111"; -- addr 003D: data 2067
when "000000111110" => romout <= "00101010100100"; -- addr 003E: data 0AA4
when "000000111111" => romout <= "01100100000011"; -- addr 003F: data 1903
when "0000001000000" => romout <= "00101010100101"; -- addr 0040: data 0AA5
when "0000001000001" => romout <= "10100000110011"; -- addr 0041: data 2833
when "0000001000010" => romout <= "00000110100100"; -- addr 0042: data 01A4
when "0000001000011" => romout <= "00000110100101"; -- addr 0043: data 01A5
when "0000001000100" => romout <= "00100000100100"; -- addr 0044: data 0824
when "0000001000101" => romout <= "00000010001100"; -- addr 0045: data 008C
when "0000001000110" => romout <= "00100000100101"; -- addr 0046: data 0825
when "0000001000111" => romout <= "00000010001101"; -- addr 0047: data 008D
when "0000001001000" => romout <= "11000000010011"; -- addr 0048: data 3013
when "0000001001001" => romout <= "00000010001111"; -- addr 0049: data 008F
when "0000001001010" => romout <= "11000010001000"; -- addr 004A: data 3088
when "0000001001011" => romout <= "10000000010110"; -- addr 004B: data 2016
when "0000001001100" => romout <= "01110100000011"; -- addr 004C: data 1D03
when "0000001001101" => romout <= "10100001010011"; -- addr 004D: data 2853
when "0000001001110" => romout <= "10000001110000"; -- addr 004E: data 2070
when "0000001001111" => romout <= "00101010100100"; -- addr 004F: data 0AA4
when "0000001010000" => romout <= "01100100000011"; -- addr 0050: data 1903
when "0000001010001" => romout <= "00101010100101"; -- addr 0051: data 0AA5
when "0000001010010" => romout <= "10100001000100"; -- addr 0052: data 2844
when "0000001010011" => romout <= "00000110100100"; -- addr 0053: data 01A4
when "0000001010100" => romout <= "00000110100101"; -- addr 0054: data 01A5
when "0000001010101" => romout <= "00100000100100"; -- addr 0055: data 0824
when "0000001010110" => romout <= "00000010001100"; -- addr 0056: data 008C

```

```

when "0000001010111" => romout <= "00100000100101"; -- addr 0057: data 0825
when "0000001011000" => romout <= "00000010001101"; -- addr 0058: data 008D
when "0000001011001" => romout <= "11000000010011"; -- addr 0059: data 3013
when "0000001011010" => romout <= "00000010001111"; -- addr 005A: data 008F
when "0000001011011" => romout <= "11000010001000"; -- addr 005B: data 3088
when "0000001011100" => romout <= "10000000010110"; -- addr 005C: data 2016
when "0000001011101" => romout <= "01110100000011"; -- addr 005D: data 1D03
when "0000001011110" => romout <= "10100001100100"; -- addr 005E: data 2864
when "0000001011111" => romout <= "10000001110111"; -- addr 005F: data 2077
when "0000001100000" => romout <= "00101010100100"; -- addr 0060: data 0AA4
when "0000001100001" => romout <= "01100100000011"; -- addr 0061: data 1903
when "0000001100010" => romout <= "00101010100101"; -- addr 0062: data 0AA5
when "0000001100011" => romout <= "10100001010101"; -- addr 0063: data 2855
when "0000001100100" => romout <= "00000110100100"; -- addr 0064: data 01A4
when "0000001100101" => romout <= "00000110100101"; -- addr 0065: data 01A5
when "0000001100110" => romout <= "10100000110011"; -- addr 0066: data 2833
when "0000001100111" => romout <= "01011110000110"; -- addr 0067: data 1786
when "0000001101000" => romout <= "11000001100100"; -- addr 0068: data 3064
when "0000001101001" => romout <= "10000000000001"; -- addr 0069: data 2001
when "0000001101010" => romout <= "01001110000110"; -- addr 006A: data 1386
when "0000001101011" => romout <= "11000000000001"; -- addr 006B: data 3001
when "0000001101100" => romout <= "00000010001101"; -- addr 006C: data 008D
when "0000001101101" => romout <= "11000010010000"; -- addr 006D: data 3090
when "0000001101110" => romout <= "10000000000010"; -- addr 006E: data 2002
when "0000001101111" => romout <= "00000000001000"; -- addr 006F: data 0008
when "0000001110000" => romout <= "01011110000110"; -- addr 0070: data 1786
when "0000001110001" => romout <= "11000011111010"; -- addr 0071: data 30FA
when "0000001110010" => romout <= "10000000000001"; -- addr 0072: data 2001
when "0000001110011" => romout <= "01001110000110"; -- addr 0073: data 1386
when "0000001110100" => romout <= "11000011111010"; -- addr 0074: data 30FA
when "0000001110101" => romout <= "10000000000001"; -- addr 0075: data 2001
when "0000001110110" => romout <= "00000000001000"; -- addr 0076: data 0008
when "0000001110111" => romout <= "01011110000110"; -- addr 0077: data 1786
when "0000001111000" => romout <= "11000000000001"; -- addr 0078: data 3001
when "0000001111001" => romout <= "00000010001101"; -- addr 0079: data 008D
when "0000001111010" => romout <= "11000010010000"; -- addr 007A: data 3090
when "0000001111011" => romout <= "10000000000010"; -- addr 007B: data 2002
when "0000001111100" => romout <= "01001110000110"; -- addr 007C: data 1386
when "0000001111101" => romout <= "11000001100100"; -- addr 007D: data 3064
when "0000001111110" => romout <= "10000000000001"; -- addr 007E: data 2001
when "0000001111111" => romout <= "00000000001000"; -- addr 007F: data 0008
when others => romout <= "00000000000000";
end case;
end process;

```

```
end RTL;
```